1

Resources of Microcontrollers

1.1 In this Chapter

This chapter is a presentation of the main subsystems of microcontrollers, seen as *resources*, organized according to one of the fundamental *architectures*: Von Neumann and Harvard. It also contains a description of the internal *CPU registers*, the general structure of a *peripheral interface*, and an overview of the *interrupt system.*

1.2 Microcontroller Architectures

A *microcontroller* is a structure that integrates in a single chip a microprocessor, a certain amount of memory, and a number of peripheral interfaces.

The Central Processing Unit (CPU) is connected to the other subsystems of the microcontroller by means of *the address and data buses*. Depending on how the CPU accesses the program memory, there are two possible architectures for microcontrollers, called Von Neumann, and Harvard.

Figure 1.1 shows the structure of a computer with Von Neumann architecture, where all the resources, including program memory, data memory, and I/O registers, are connected to the CPU by means of a unique address and data bus.

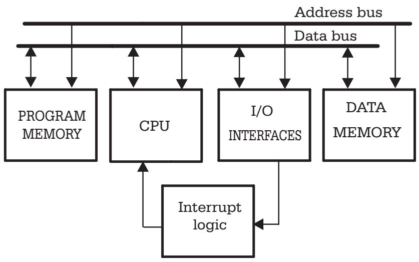

Fig. 1.1. Block diagram of Von Neumann architecture

A typical microcontroller having Von Neumann architecture is 68HC11 from Motorola. In HC11, all resources are identified by unique addresses in the same address space, and can be accessed using the same instructions. For example, in case of the instruction:

```
LDAA     <address>     ;load accumulator a from <address>
```

the operand indicated by the label <address> can be any of the microcontroller's resources, from I/O ports, to ROM constants. This way of accessing resources allows the existence of complex instructions like this:

```
ASL      35,x          ;arithmetic shift left the memory
                       ;location with the address
                       ;obtained by adding 35 to the
                       ;index register X.
```

Therefore, the Von Neumann microcontrollers tend to have a large instruction set, including some really complex instructions. This is the reason why computers having the Von Neumann architecture are often called CISC, or Complex Instruction Set Computers.

The main disadvantage of this architecture is that the more complex the instruction, the longer it takes to fetch, decode, execute it, and store the result. The instruction in the above example takes six machine cycles to execute, while the instruction for integer divide, IDIV, needs no less than 41 machine cycles to complete.

The Harvard architecture was created to increase the overall speed of computers in the early years, when very slow magnetic core memory was used to store the program. It includes an additional, separate bus to access the program memory (refer to Fig. 1.2).

The presence of the second bus makes the following things possible:

- While an instruction is executed, the next instruction can be fetched from the program memory. This technique is called *pipelining* and brings a significant increase of computer speed.
- The program memory can be organized in words of different size from, and usually larger than, the data memory. Wider instructions mean a greater data flow to the CPU, and therefore the overall speed is higher.

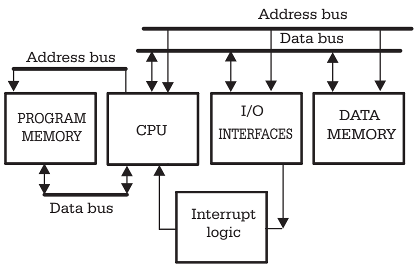

Fig. 1.2. Block diagram of Harvard architecture

Such architecture, along with reducing and optimizing the instruction set, mean that most instructions execute in a single machine cycle. Since the Harvard architecture is often accompanied by the reduction of the size and complexity of the instruction set, computers with this architecture are also called Reduced Instruction Set Computers (RISC). For example, some PIC microcontrollers have an instruction set of only 35 instructions, compared to more than 100 for HC11. The speed increase is even higher.

The separate bus for the program memory makes the access of the program to constants (such as tables, strings, etc.) located in ROM more complicated and more restrictive. For example, some PIC microcontrollers have the program memory organized in 14-bit wide *words*, which makes locating and accessing a constant presented as a *byte* possible only by embedding the constant in a special instruction. For this purpose, the instruction "RETLW k" (Return from subprogram with constant k in register W) has been provided.

The AVR microcontrollers have the program memory organized into 16-bit words, which makes the task of accessing constants in program memory easier, because each 16-bit word can store two 8-bit constants. A special instruction LPM (Load from Program Memory) allows access to ROM constants.

1.3 The Memory Map

From the programmer's point of view, a microcontroller is a set of resources. Each resource is identified by one or more *addresses* in an *address space*. For example, the 68HC11E9 microcontroller has its internal RAM memory organized as 512 locations, having addresses in the range \$0000–\$01FF, the ROM memory occupies the addresses in the range \$D000–\$FFFF (12288 locations), and the I/O register block takes the address form \$1000–\$103F (64 locations).

The *memory map* is a graphic representation of how the resources are associated with addresses (see Fig. 1.3 for an example of a memory map).

Obviously, not all addresses are related to existing resources – in some cases it is possible to add external memory or I/O devices, to which we must allocate distinct addresses in the address space.

Normally, the memory map is determined by the hardware structure formed by the microcontroller and the external devices (if any), and cannot be dynamically modified during the execution of a program.

However, there are situations when, by writing into some special configuration registers, the user can disable resources (such as the internal ROM memory, or the EEPROM memory) or can relocate resources in a different area of the address space. But even in these cases, the access to the configuration registers is restricted, and the modification becomes effective after the next RESET.

Figures 1.3 and 1.4 show the memory maps for a microcontroller with Von Neumann architecture, MC68HC11E9, operating in single-chip mode, and for a RISC microcontroller, the AVR AT90S8535.

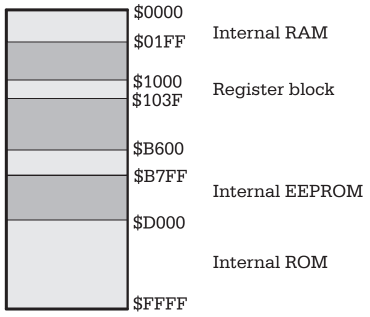

Fig. 1.3. Memory map for 68HC11E9 operating in single-chip mode

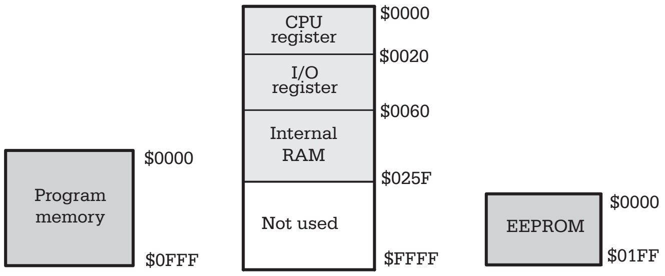

Fig. 1.4. Memory map for AT90S8515 operating in single-chip mode

Note, for the AVR microcontroller, the presence of three different address spaces, one for data memory and I/O registers, and two more for the program memory and the EEPROM.

The 8051 microcontrollers are considered to belong to the Harvard architecture, but they are CISC, and do not allow pipelining; therefore they look more like Von Neumann computers with the capability to access program memory, and data memory as different *pages*. The two distinct memory pages are accessed through *the same physical bus*, at different moments time in. Fig. 1.5 shows the memory map for an 8051 MCU operating in single-chip mode. There are two address spaces here too, one for the program memory and the other for data memory and special function registers.

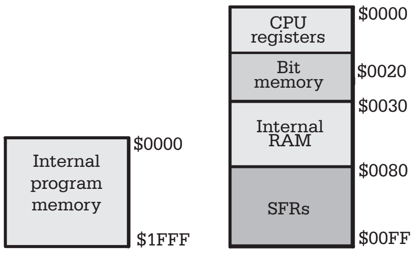

Fig. 1.5. Memory map for 8051 operating in single-chip mode

One unique feature of this microcontroller is the presence of a RAM area, located in the address range \$0020–\$002F, which is bit addressable, using special instructions. This artifice allows the release of RAM memory by assigning some Boolean variables to individual bits in this area, rather than using a whole byte for each variable, because 8051 is low on this resource: only 80 RAM locations are available for variables and stack.

Standard 8051 microcontrollers do not have internal EEPROM memory.

1.4 CPU Registers

The good thing about CPU registers is that they are part of the CPU, and an operand located in these registers is immediately available as input to the arithmetic and logic unit (ALU). Since the instructions having operands in the registers of the CPU are executed faster, the microcontrollers designed for higher speed tend to have more internal registers. While HC11 has only two accumulator registers, the AVR family has as many as 32 such registers.

1.4.1 The CPU Registers of HC11

HC11 has seven internal registers, plus the CPU status register, called the Condition Code Register (CCR).

The accumulator registers A and B are general-purpose 8-bit registers. They can be concatenated to form a 16-bit register called D, where A is the most significant byte, and B is the least significant byte. This feature creates a remarkable flexibility for 16-bit arithmetic operations.

The index registers X and Y are 16-bit registers, which can also be used as storage registers, 16-bit counters; and most important, they can store a 16-bit value, which, added with an 8-bit value contained in the instruction itself, form the effective address of the operand when using the indexed addressing mode.

The Stack Pointer (SP) register is a 16-bit register, that must be initialized by software with the *ending address* of a RAM memory area, called the *stack*. SP automatically decrements each time a byte is pushed to the stack, and increments when a byte is pulled from stack. Thus, SP always points to the first free location of the stack. The stack is affected in the following situations:

- During the execution of the instructions BSR, JSR (Branch or Jump to Subroutine), the return address is automatically pushed on to the stack and the SP is adjusted accordingly. The instruction RTS (Return from Subroutine) pulls this value from the stack and reloads it into the program counter.
- During the execution of push and pull type instructions, used to save and restore the contents of the CPU registers to the stack.
- During the execution of an interrupt, and when returning from an interrupt service routine upon the execution of the RTI (Return from Interrupt) instruction.

SP may be directly accessed by means of the LDS (load SP) and STS (Store SP) instructions or indirectly, using transfer instructions like TXS, TYS (Transfer X/Y to SP) or TSX, TSY (Transfer SP to X/Y).

The Program Counter (PC) register is a 16-bit register, that contains the address of the instruction following the instruction currently executed.

The Condition Code Register (CCR) is an 8-bit register with the following structure:

CCR	7	6	5	4	3	2	1	0
	S	X	H	I	N	Z	V	C
RESET	1	1	0	1	0	0	0	0

The bits C (Carry/Borrow), V (Overflow), Z (Zero), N (Negative) and H (Half Carry) are status bits, set or cleared according to the result of the arithmetic and logic instructions. Refer to the data sheet for details on how these bits are affected by each instruction.

The bits I (General Interrupt Mask), X (XIRQ Interrupt Mask), and S (Stop disable) are control bits used to enable/disable the interrupts, or the low-power operating mode. When I = 1 all maskable interrupts are disabled. X = 1 disables the non-maskable interrupt XIRQ, and S = 1 blocks the execution on the STOP instruction, which is treated like a NOP.

Some CCR bits (C, V, I) can be directly controlled by means of the instructions SEC (Set Carry), CLC (Clear Carry), SEV (Set Overflow Bit), CLV (Clear Overflow Bit), SEI (Set Interrupt Mask), and CLI (Clear Interrupt Mask). The CCR as a whole may be read or written using the instructions TPA (Transfer CCR to A) and TAP (Transfer A to CCR)

1.4.2 The CPU Registers of AVR

The CPU of the AVR microcontrollers has 32 general-purpose registers, called R0–R31. The register pairs R26–R27, R28–R29, R30–R31 can be concatenated to form the X, Y, Z , registers, which can be used for indirect addressing (R26 is XL – lower byte of X, R27 is XH – higher byte of X, R28 is YL, R29 is YH, R30 is ZL and R31 is ZH). The registers R16–R31 may be the destination of immediate addressed operands like LDI (Load Register Immediate) or CPI (Compare Immediate). *Unlike HC11, the CPU registers of AVR are present with distinct addresses in the memory map.*

The Program Counter (PC) has functions similar to those of the PC register of HC11. The difference is that the size of PC is not 16 bits, and is limited to the length required to address the program memory (in case of AT90S8515 only 12 bits are needed to address the 4K of program memory). PC is cleared at RESET.

The Stack Pointer (SP) has 16 bits, and is placed in the I/O register address space, which makes it accessible to the programmer only by means of the IN and OUT instructions, as two 8-bit registers SPH, and SPL.

The CPU status register is called SREG and has the following structure:

SREG	7	6	5	4	3	2	1	0
	I	T	H	S	V	N	Z	C
RESET	0	0	0	0	0	0	0	0

The meaning of the bits in SREG is slightly different from those of HC11:

The I bit – Global Interrupt Enable/Disable Bit – has an opposite action: when set to 1 the interrupts are enabled. The instructions that control this bit have the same mnemonic SEI (Set I bit) and CLI (Clear I bit).

T – Bit Copy Storage. The status of this bit can be modified by the instructions BST (Bit Store) and BLD (Bit Load), thus allowing the program to save the status of a specific bit from a general-purpose register, or transfer this information to a bit from another register. There is also a pair of conditional branch instructions which test this bit: BRTS (Branch if T bit is Set), and BRTC (Branch if T bit is Clear)

S –Sign Bit – It is the exclusive OR between N and V

The other bits in SREG (C, Z, N, V, H) have the same meaning described for HC11. The AVR microcontrollers have distinct SET–CLEAR instructions for each of the SREG bits.

1.4.3 The CPU Registers of 8051

The accumulator A is a general-purpose 8-bit register, used to store operands or results in more than a half of the instruction set of 8051.

The R0–R7 registers are 8-bit registers, similar to the registers R0–R31, described for the AVR family of microcontrollers. There are four sets (or *banks*) of such registers, selected by writing the bits [RS1:RS0] in the CPU status register PSW, described below.

The four sets of eight registers each occupy 32 addresses in the address space of data memory, at the addresses [0000h–0007h], [0008h–000Fh], [0010h–0017h], [0018h–001Fh] (refer to Fig. 1.4).

The accumulator B is another general-purpose 8-bit register, having functions similar to the R0–R7 registers. Besides that, the accumulator B is used to store one of the operands in the case of the arithmetic instructions MUL AB and DIV AB.

The Data Pointer Register (DPTR) is a 16-bit register, used for indirect addressing of operands, in a similar way to the X, Y, Z registers of AVR.

The Program Counter (PC) is a 16-bit register similar to the PC of HC11. PC is cleared at RESET, thus all programs start at the address 0000h.

The Stack Pointer (SP) has the following distinctive features, compared to HC11 and AVR:

- It is an 8-bit register, i.e. it can address a memory area of 256 bytes maximum. 8051 can only use the internal memory for the stack.

- Unlike HC11 and AVR where SP is initialized with an address at the end of RAM, and decrements with each byte pushed on to the stack, the SP of 8051 increments when data is added to the stack.
- For HC11 and AVR, SP points to the first free byte of the stack area. The SP of 8051 indicates the last occupied location of the stack. At RESET, SP is automatically initialized with 07h, hence the first byte pushed to the stack will occupy the location with the address 08h.

The Processor Status Word (PSW) is similar to CCR of HC11 or SREG of AVR, and has the following structure:

PSW	7	6	5	4	3	2	1	0
	CY	AC	F0	RS1	RS0	OV	–	P
RESET	0	0	0	0	0	0	0	0

The bits CY, AC and OV have similar functions to the bits C, H, and V of HC11 and AVR.

[RS1:RS0] – Register bank select bits

P – Parity bit. P = 1 if the accumulator contains an odd number of 1s, and P = 0 if the accumulator contains an even number of 1s. Thus the number of 1s in the accumulator plus P is always even. The bits PSW1 and PSW5 (F0) are uncommitted and may be used as general-purpose status flags.

1.5 The Peripheral Interfaces

Microcontrollers are designed to be embedded in larger systems, and therefore they must be able to interact with the outside world. This interaction is possible by means of the peripheral interfaces. The general structure of a peripheral interface is shown in Fig. 1.6.

Depending on the complexity of the specific circuits to be controlled by the program, any peripheral interface contains one or more control and status registers, and one or more data registers. These registers are normally located in the address space of the data memory, and are accessed as RAM locations.

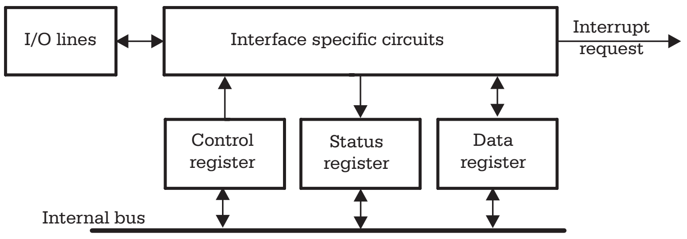

Fig. 1.6. Typical structure of a peripheral interface

The most common peripheral interfaces, present in almost all the usual microcontrollers, are:

- The I/O (Input/Output) ports.
- The asynchronous serial interface (SCI, UART)
- The synchronous serial interface (SPI)
- Several types of timers
- The analog to digital (A/D) converters

The following chapters contain detailed descriptions of each of the above peripheral interfaces. Most of the peripheral interfaces have a common feature, which is the capability to generate *interrupt requests* to the CPU, when some specific events occur. This feature is analyzed in the next paragraph.

1.6 The Interrupt System

1.6.1 General Description of the Interrupt System

Most of the events related to the peripheral interfaces, like the change of status of an input line, or reception of a character on the serial communication line, are asynchronous to the program running on the CPU. There are two possible ways to inform the CPU about these events:

- One solution is to write the program so that it periodically tests the status of some flags associated with the external events. This technique is called *polling*.
- The other solution is to interrupt the main program and execute a special subroutine when the external event occurs.

An *interrupt* is a mechanism that allows an external event to temporarily put on hold the normal execution of the program, forcing the execution of a specific subroutine. Once the interrupt service subroutine completes, the main program continues from the point where it was interrupted.

At the CPU level, this mechanism involves the following steps:

1. The identification of the interrupt source. This is automatically done by hardware.
2. Saving the current value of the PC register, thus providing a means to return from the interrupt service routine. The contents of PC are saved to the stack, and the operation is also done by hardware.
3. Then, the PC is loaded either with, or from, the address of a reserved memory area, called the *interrupt vector*. For each possible interrupt, a unique vector is assigned. *The interrupt vectors are hardwired and cannot be modified by the user.*
4. At the address of the interrupt vector, the program must contain either the address of the interrupt service routine (HC11 uses this technique) or an instruction for an unconditional jump to this routine (AVR and 8051 work this way).
5. The next step is the execution of the *Interrupt Service Routine (ISR)*. This is a program sequence similar to a subroutine, but ending with a special instruction

called Return from Interrupt (RTI, RETI). To make sure that the main program is continued exactly from the status it had in the moment when the interrupt occurred, it is crucial that all the CPU registers used by the interrupt service routine are saved at the beginning of the ISR, and restored before returning to the main program. Some microcontrollers, like the HC11 family, are provided with a hardware mechanism to save the whole CPU status, upon reception of an interrupt request. The status is restored by the instruction RTI (Return from Interrupt) before the actual return to the main program. *In all other cases, it is the user's responsibility to save and restore the CPU status in the interrupt service routine.*

6. The final step in handling an interrupt is the actual return to the main program. This is done by executing a RTI (RETI) instruction as mentioned before. When this instruction is encountered, the contents of PC, saved in step 2, are retrieved from the stack and restored, which is equivalent to a jump to the point where the program was interrupted. The process of returning from an ISR is similar to returning from a regular subroutine, but there is an important difference: the interrupt service routines cannot be interrupted, and therefore once an interrupt has been acknowledged, further interrupts are automatically disabled. They are re-enabled by the RTI (RETI) instruction. All interrupts occurring during the execution of an ISR are queued and will be handled one by one, once the ISR is serviced.

Important note. The stack is essential for the interrupt system. Both the PC and the CPU status are saved in the stack when handling interrupts. Therefore, the SP must be initialized by software before enabling the interrupts.

The interrupt service routine *must* save the CPU status and restore it before returning to the main program.

If two or more interrupt requests occur simultaneously, they are serviced in a predetermined order according to a hardwired priority. Refer to the data sheet for each microcontroller for details.

The software control over the interrupt system is exerted either globally, by enabling/disabling all the interrupts by means of specific instructions, or individually, by setting or clearing some control bits, called *interrupt masks*, associated with each interrupt. In other words, the process of generating an interrupt request is double conditioned, as shown in Fig. 1.7.

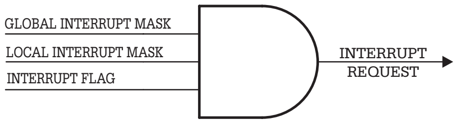

Fig. 1.7. Double conditioning of interrupt requests

The INTERRUPT FLAG is the actual interrupt source, and, usually, is a flip-flop set by the external event. This bit is, in most cases, accessible for the program as a distinct bit in the status register of the peripheral interface.

The LOCAL INTERRUPT MASKS are control bits, located in the control registers of the interface. When set to 1 by software, the interrupts from that specific interface are enabled.

The GLOBAL INTERRUPT MASK is a bit located in the CPU status register (CCR, SREG, PSW) that enables or disables all interrupts.

In some cases, it is required that the CPU is informed immediately about certain important internal or external events, regardless of the status of the global interrupt mask. The solution to this problem is the *non-maskable interrupt*, which is unconditionally transmitted to the CPU. A special case of non-maskable interrupt can be considered the RESET. Basically, the behavior of the MCU at RESET is entirely similar to the process of identification and execution of a non-maskable interrupt.

1.6.2 Distinctive Features of the Interrupt System of HC11

The most important feature of the interrupt system of HC11 is that the CPU status is automatically saved by hardware, right after the PC is saved. This feature simplifies the programmer's work, but it wastes time saving and restoring *all* CPU registers. In most cases, the interrupt service routine does not use *all* the CPU registers, but needs to be executed as fast as possible.

The global control of the interrupt system is performed by means of the I bit in the CCR register. When $I = 1$, all maskable interrupts are disabled. When $I = 0$, the interrupts coming from a specific peripheral interface are enabled if the local mask associated with that interface is set to 1. The I bit can be controlled by means of the instructions SEI (Set Interrupt Mask) equivalent to Disable Interrupts, and CLI (Clear Interrupt Mask), equivalent to Enable Interrupts.

Besides the maskable interrupts, HC11 has three non-maskable interrupts, without counting the three possible RESET conditions (activation of the external RESET line, clock monitor fail reset, and watchdog reset). These are: XIRQ – External non-maskable interrupt, ILLOP – Illegal opcode trap, and SWI – software interrupt.

The XIRQ interrupt is generated upon detection of a logic level LOW on the XIRQ input line, if the X bit in CCR is clear. The X bit acts similarly to I, but it only affects the XIRQ interrupt, and it is not affected by the SEI and CLI instructions. The only way the user can alter the status of this bit is by the TAP (Transfer A to CCR) instruction.

The illegal opcode trap is an internal interrupt generated when an unknown opcode is fetched and decoded into the CPU.

A software interrupt is generated when the instruction SWI is decoded. This is useful for defining breakpoints in a program for debug purposes.

The priority of the interrupts is hardwired. However, it is possible to define one of the interrupts as the highest priority non-maskable interrupt. For this purpose, the bits [PSEL3–PSEL0] (Priority Select bits) in register HPRIO (Highest Priority Interrupt Register) code the interrupt with the highest priority.

The vector area for HC11 is located at the end of the address space between the addresses $FFC0–$FFFF. See the data sheets for the list of exact addresses assigned to each interrupt vector.

1.6.3 Distinctive Features of the Interrupt System of AVR

There are a few differences between the interrupt system of AT90S8535 and that of HC11. They are listed below:

- The interrupt vector does not contain *the address* of the interrupt service routine, but a jump instruction to that routine.
- The vector area is located at the beginning of the program memory address space, between the addresses $0000 and $0010.
- There are no non-maskable interrupts besides RESET.
- The I bit in SREG acts differently, compared to HC11: when $I = 1$, the interrupts are enabled.
- There is no equivalent to the HPRIO register, and no other means to modify the hardwired relative priority of interrupts.

1.6.4 Distinctive Features of the Interrupt System of 8051

8051 has only five possible interrupt sources, compared to 16 for AVR, and 18 for HC11. The vectors are placed at the beginning of the program memory address space and must be initialized by the software to contain a jump to the interrupt service routine.

The interrupts are enabled and disabled according to the same principles described for HC11 and AVR. The difference is that all the control bits associated with the interrupt system are placed in a Special Function Register (SFR) called IE (Interrupt Enable register) located at the address A8h. This register contains the global interrupt control bit, called in this case EA (Enable All interrupts), and bits to enable each individual interrupt.

One interesting distinctive feature of the interrupt system of 8051 is the possibility to choose between two priority levels (low and high) for each interrupt. To this purpose, a special register called IP (Interrupt Priority register) contains a bit associated with each interrupt. When the priority bit is 0, the associated interrupt has a low priority level, and when the priority bit is 1, the interrupt has high priority. *Unlike HC11 and AVR, for 8051 a high-priority interrupt can interrupt a low-priority interrupt service routine.*

8051 does not save the CPU status automatically, therefore the interrupt service routine *must* save and restore the registers used, including PSW.

1.7 Expanding the Resaurces of Microcontrollers

In many cases it is possible that the internal resources of a microcontroller are insufficient for certain applications. A typical example is when the number of variables used to store data exceeds the capacity of the internal RAM memory. The obvious solution to these situations is to add external components by creating an *expanded microcontroller structure*.

The disadvantage of this solution is that a significant number of the available I/O lines are used to create the external bus for accessing the new resources, and are no longer available for normal I/O operations. Note that not all microcontrollers can operate with an external bus. The following paragraphs describe how to create and use expanded microcontroller structures.

1.7.1 HC11 Operating with External Bus

The HC11 microcontrollers have two pins, called MODA and MODB, which control the operating mode. At RESET the status of these pins is read and, according to the result, the microcontroller selects one of the operating modes listed in Table 1.1.

As shown in Table 1.1, there are four possible operating modes, among which two are *special* and the other two are *normal*. In the *special bootstrap* mode, a small ROM memory area becomes visible in the memory map. This ROM contains a short program, called a bootloader, which is executed after RESET, allowing the user to load and run a program in the internal RAM. This is useful, for example, to program the internal ROM memory. See App. A.5 for details on how to do this.

The special test operating mode is destined for factory testing, and will not be discussed in this book.

In the expanded operating mode, some of the MCU I/O lines are used to implement the external bus. In some cases, to reduce the number of I/O lines used for this purpose, the external data bus is multiplexed with the lower byte of the address bus.

Demultiplexing requires an external latch, and a signal AS (Address Strobe), generated by the MCU, as shown in Fig. 1.8.

Besides AS, the MCU generates the signal R/W\ (Read/Write) to specify the direction of the transfer on the data bus. A detailed example of using HC11 in expanded mode is presented in App. A.4.

From the programmer's point of view, when using HC11 in expanded operating mode, the following details must be considered:

- All the MCU resources, except the I/O lines used to implement the external bus, are still available, and have the same addresses.
- The internal ROM can be disabled by clearing the ROMON bit in the CONFIG register.

Table 1.1. Selection of the operating mode of HC11

MODB	MODA	Operating mode
0	0	Special bootstrap
0	1	Special test
1	0	Single chip
1	1	Expanded

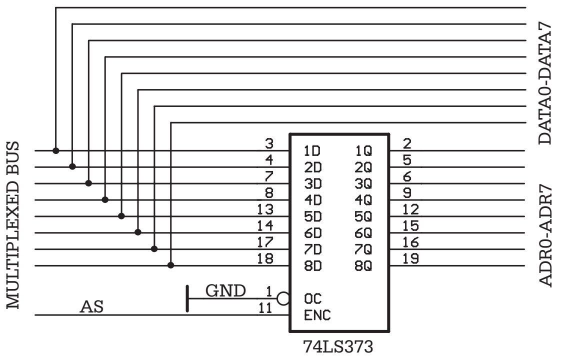

Fig. 1.8. Demultiplexing the external bus

1.7.2 AT90S8515 Operating with External Bus

From the two distinct buses of the AVR microcontrollers, only the bus used for the data memory can be, *in some cases*, expanded to connect external RAM, or RAM – like external devices. AT90S8515 uses the lines of port A to multiplex the data bus with the lower order address bus, and port C for the high order address bus. The signal that strobes the address into the external latch is called ALE (Address Latch Enable). The direction of the transfer is indicated by two signals RD\ (Read) and WR\ (Write) – both active LOW. The circuit for demultiplexing the bus is identical to the one used by HC11, shown in Fig. 1.8.

The external bus is activated by software, writing 1 to the SRE bit (Static RAM Enable) in the register MCUCR (MCU Control Register). When using slower external memory, there is the possibility of including a WAIT cycle to each access to the external RAM. Writing 1 to the SRW (Static RAM Wait) bit of MCUCR enables this feature.

Appendix A.8 shows an example on how to use AT90S8515 with an external bus.

1.7.3 8051 Operating with External Bus

The external bus of 8051 is also multiplexed. Port P0 is used for the data bus and the low-order address bus, and port P2 implements the high-order address bus. The strobe signal for demultiplexing the bus is called ALE (Address latch Enable).

What is specific for 8051 is the capability to access on the same physical bus two pages of external memory: one for program memory, and the other for data memory. The two types of access are distinguished by means of a special signal, called PSEN\ (Program Store Enable), generated by the MCU, active LOW. When PSEN\ is active, the external logic must select an external ROM memory in order to provide program code to the MCU through the data bus. The access to the external RAM is controlled at the hardware level by the signals RD\ (Read) and WR\ (Write), generated by the microcontroller. An additional signal, called EA\ (External Access), active LOW, disables the internal ROM, and redirects all the accesses to the internal program memory (in the address range 0000h–0FFFh) to the external bus. Any access to program memory at addresses higher that 1000h are directed to the external bus regardless of the status of the EA\ input.

At the software level, accessing the external RAM is done by means of the special instruction MOVX (Move to or from external RAM).

Chapter 11 and App. A.12 contain detailed examples of using 8051 with an external bus.

1.8 Exercises

SX 1.1

Write a fragment of program to access a constant stored in the program memory of a MCU with the Von Neumann architecture (HC11).

Solution

Like all Von Neumann computers, HC11 uses a single bus to connect all the resources to the CPU, and has a single address space. Therefore, there is no difference between the ways it accesses constants located in the program memory or variables stored in RAM. The following code fragment does the job:

```
          .....
          LDAA      ROMTAB
          .....
ROMTAB    DB        $55        ;This defines $55 as
                               ;a constant located
                               ;in the program memory area
```

After the execution of the instruction LDAA ROMTAB, the contents of the accumulator A are identical to the contents of the memory location having the address ROMTAB ($55 in this case). The following example uses the X register as a pointer to a table of constants, ROMTAB. After the execution of this code, A contains $55, B equals $AA, and X contains the address ROMTAB.

```
          .....
          LDX       #ROMTAB    ;set X to point to ROMTAB
          LDAA      0,X        ;read one constant into A
          LDAB      1,X        ;read another constant into B
          .....
ROMTAB    DB        $55        ;This defines $55 as a
                               ;constant located in the
                               ;program memory area
          DB        $AA
```

SX 1.2

Write a code fragment to access a constant located in the program memory of a microcontroller having the Harvard architecture (AT90S8515 AVR)

Solution

The AVR microcontrollers use the special instruction LPM to access constants stored in the program memory. LPM copies to R0 the contents of the program memory location with the address specified by the Z register. The problem is that AVR microcontrollers have the program memory organized in 16-bit words and the assembler assigns a word address to any label found in the code section.

LPM interprets the contents of the register Z in the following way: the most significant 15 bits of Z represent the address of the 16-bit word, and the least significant bit is the address of the byte within the 16-bit word: LSB(Z) = 0 indicates the least significant byte, and when LSB(Z) = 1 the most significant byte is addressed.

```
          ....
          LDI       ZH,high(ROMTAB<<1)
          LDI       ZL,low(ROMTAB<<1)
          LPM                      ;read first byte
          MOV       R1,R0          ;save $34 to R1
          ADIW      ZL,1           ;increment Z
          LPM                      ;read second byte
          MOV       R2,R0          ;save $12 to R2
          ....
ROMTAB    DW        $1234
```

In the above example, the expression `(ROMTAB<<1)` means ROMTAB shifted one position to the left, and `high(ROMTAB<<1)` designates the most significant byte of the 16-bit value `(ROMTAB<<1)`. The first LPM reads in R0 the lower byte of the constant located at the address ROMTAB ($34) and, after incrementing the address, (ADIW ZL,1) the second LPM reads the most significant byte of the constant ($12).

SX 1.3

Write a code fragment to access a constant stored in the program memory of an 8051 microcontroller.

Solution

8051 uses a special instruction to read ROM constants. This is MOVC (Move Code) and has the following syntax:

```
   MOVC A,@A+DPTR
```

The effect of this instruction is that the accumulator A is loaded with the contents of the program memory location having the address obtained by adding the 16-bit integer in DPTR with the unsigned byte in A.

```
          CLR       A             ;clear accumulator A
          MOV       DPTR,#ROMTAB  ;load DPTR with the address
                                  ;ROMTAB
          MOVC      A,@A+DPTR     ;get constant in A
          .....
          .....
ROMTAB:
          DB        55h           ;define the constant here
          .....
```

SX 1.4

Provide an example of interrupt vector initialization for HC11.

Solution

The HC11 interrupt vector is a 2-byte memory space, which must be initialized with the starting address of the interrupt service routine. For example, the interrupt vector associated with RESET (remember RESET is treated as a non-maskable interrupt) is located at the addresses \$FFFE–\$FFFF. At RESET, the MCU reads the two bytes from these addresses (\$FFFE contains the most significant byte) and loads the resulting 16-bit value into PC.

```
MAIN      .......                 ;Program entry point at
                                  ;RESET
          .......
          ORG       $FFFE         ;store the value of the
                                  ;label MAIN at $FFFE-$FFFF
          DW        MAIN
```

SX 1.5

Provide an example of interrupt vector initialization for AVR.

Solution

The vector area of the AVR microcontrollers start at the address \$0000 in the address space of the program memory. Each interrupt vector occupies a 16-bit word. Unlike HC11, the AVR simply loads the hardwired value of the vector into the PC when an interrupt is acknowledged. Initializing the vector consists in placing in the address of the vector an instruction of an unconditional jump to the interrupt service routine. For example, the interrupt vector associated with the analog comparator has the address $000C. The initialization of this vector is shown below:

```
          .......
          .ORG      $000C
          RJMP      ANA_COMP      ;unconditional jump to the
                                  ;interrupt handler
          .......
ANA_COMP:
          .......
          RETI
```